Node.js is a popular choice for teams that need to design, build, test, deploy, maintain, and monitor large-scale distributed systems. Starting with a detailed overview of the Node.js architecture, this book covers topics that will help in application development, testing, deployment, and maintenance.

You will learn about concurrency, event loops, callbacks and streams. Furthermore, step-by-step instructions on deploying applications to providers such as DigitalOcean and Heroku will be provided, including information on setting up load balancers and proxies. Message queues and other techniques for managing state and session data at scale will also be covered.

A series of examples on deploying your Node.js applications in production environments are provided, including a discussion on setting up continuous deployment and integration for your team. Popular tools for testing, deploying, building, and monitoring Node.js applications are covered, helping you get up and running quickly.

What you will learn from this book

- Learn the strengths of Node.js and how to optimize your code to make it ready for deployment into production
- Use Docker and Vagrant to create many virtual instances of your Node.js applications
- Deploy Node.js repositories to the cloud using Heroku, OpenShift, and DigitalOcean
- Utilize native Node.js modules, or NGINX, to load balance your applications
- Optimize your Node runtime by tweaking V8, managing memory intelligently, and using Redis to manage state data
- Discover how to use Gulp, Browserify, npm, Mocha, Chai, Sinon, and other tools to simplify your build/test/release process
- Manage your production deployments with Git, Jenkins, and Ansible

Who this book is written for

If you are an intermediate or advanced developer deploying your Node.js applications, then this book is for you. If you have already built a Node.js application or module and want to take your knowledge to the next level, this book will help you find your way.


$ 44.99 US
£ 29.99 UK

Sandro Pasquali

Learn how to build, test, deploy, monitor, and maintain your Node.js applications at scale
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Managing Memory and Space'
- A synopsis of the book’s content
- More information on Deploying Node.js
About the Author

**Sandro Pasquali**, in 1997, formed Simple.com, a technology company that sold the world's first JavaScript-based application development framework and was awarded several patents for deployment and advertising technologies that anticipated the future of Internet-based software. Node represents, for him, the natural next step in the inexorable march toward the day when JavaScript powers nearly every level of software development.

Sandro has led the design of enterprise-grade applications for some of the largest companies in the world, including Nintendo, Major League Baseball, Bang and Olufsen, LimeWire, AppNexus, Conde Nast, and others. He has displayed interactive media exhibits during the Venice Biennial, won design awards, built knowledge management tools for research institutes and schools, and started and run several start-ups. Always seeking new ways to blend design excellence and technical innovation, he has made significant contributions across all levels of software architecture, from data management and storage tools to innovative user interfaces and frameworks.

He is the author of *Mastering Node.js*, also by Packt Publishing, which takes you on a deep dive into Node, teaching you how to use it to build modern, fast, and scalable networked software.

Sandro runs a software development company in New York and trains corporate development teams interested in using Node and JavaScript to improve their products. He spends the rest of his time entertaining his beautiful daughter and his wife.
Over the past few years, Node.js has found its way into the technology stack of Fortune 500 companies, mobile-first start-ups, successful Internet-based businesses, and other enterprises. In addition to validating its power as a platform, this success has exposed a shortage of tools for managing, deploying, and monitoring Node.js applications. Even though the Node.js community is open and collaborative, comprehensive information on how professional Node developers design, test, and push their code into production environments is hard to find.

This book is an attempt to close that gap in knowledge by explaining and documenting techniques and tools that working Node.js developers can use to create scalable, smart, robust, and maintainable software for the enterprise.

After a brief introduction to Node and the philosophy behind its design, you will learn how to install and update your applications on local servers and across the cloud. This foundation is then built upon by proceeding step by step through load balancing and other scaling techniques, explaining how to handle sudden changes in traffic volume and shape, implement version control, and design memory-efficient, stateful, distributed applications.

Once you've completed the groundwork essential to creating production-ready systems, you will need to test and deploy them. Filled with real-world code examples, this book is structured as a progressive workbook explaining strategies for success to be used during each stage of the testing, deploying, monitoring, and maintaining marathon that every successful piece of software runs.

When you've finished this book, you will have learned a set of reusable patterns directly applicable to solving the problems that you are dealing with today and will have laid the foundation to make confident decisions about how to build and deploy your next project.
What this book covers

Chapter 1, Appreciating Node, goes into the thinking behind Node.js, helping you to think clearly about what Node.js is good at, what it is less good at, and how to leverage Node.js to solve real-world challenges.

Chapter 2, Installing and Virtualizing Node Servers, teaches you how to create a basic Node.js application and get it running on a server. You will also learn how to do the same thing on certain popular cloud-hosting providers, such as DigitalOcean and Heroku, additionally learning how to use Docker to create lightweight, easily replicable, virtual machines.

Chapter 3, Scaling Node, explores both vertical and horizontal scaling techniques. You will learn how to use the cluster module to maximize Node’s effectiveness on a single server and how to coordinate many distributed Node.js servers to handle increasing network traffic, learning about Nginx load balancing, setting up proxies using message queues, and coordinating interprocess communication in the process.

Chapter 4, Managing Memory and Space, demonstrates how good engineering practices never go out of style. We start with a discussion of microservices, introducing general techniques to design systems composed of small, focused, communicating processes. Next comes a deep exploration of how to optimize JavaScript, particularly for the V8 compiler, beginning with several examples of memory-efficient ways to store and retrieve event data in Redis, we look at caching strategies, session management, and using CDNs to reduce server load.

Chapter 5, Monitoring Applications, explains strategies to effectively monitor your application once it has been deployed. Examples of using various third-party monitoring tools are included as well as examples outlining how you can build your own custom system sampling and logging modules. Finally, we look at debugging techniques, examining several tools and strategies to help you find and prevent runtime bottlenecks.

Chapter 6, Building and Testing, introduces certain considerations when creating a build pipeline for your application. Full examples of using Gulp, Browserify, and npm to create build tools are provided as well as information on testing with Mocha, mocking with Sinon, and using PhantomJS for headless browser testing.

Chapter 7, Deploying and Maintaining, walks you through the entire deployment pipeline, from setting up virtualized development environments to building continuous integration into your workflow. You will learn about using GitHub webhooks and Vagrant and using Jenkins to automate your deployment process. Additionally, the npm package manager will be fully dissected, and strategies for dependency management will be discussed.
Today's developer has easy access to surprisingly inexpensive storage solutions. The movement away from monolithic systems toward composed and distributed ones has certain advantages, yet inevitably introduces a few new problems. The availability of cheap storage should not be an excuse to push everything you can into memory or onto a disk without any limit, for instance. Also, where does the state reside in such a system? Does a cluster of servers share a common database connection? How is data synchronized in such a setup? If you are using a *shared-nothing noSQL* architecture, how are state changes communicated across all actors?

There are many considerations. Always seeking to use a minimum of resources is a good guiding principle. In this chapter, we will look at ways to reduce the cost of data storage in your Node programs, including tips on writing efficient, optimized code. Certain strategies for efficiently sharing data across distributed servers will be discussed, including caching strategies, microservices, interprocess messaging, and other techniques to keep your systems fast, light, and scalable. Examples demonstrating how to use tokens to manage user session data efficiently at scale and storing extensive user activity data compactly using Redis will help you put these ideas into practice.
Dealing with large crowds
Because Node is designed to make the writing of networked applications easier, those using Node are often building applications composed of many isolated services that are connected via message queues, sockets, REST APIs, and so on. I will describe these as distributed applications composed of isolated services coupled and coordinated through a network into systems that appear integrated to clients. In this section and the sections that follow, we will consider how isolated services can be designed to be memory efficient with a small footprint.

For the purposes of this section and what follows, the word **microservice** will be used when referring to application architectures composed of many small cooperating services. Generally, we'll explore ideas around how well-designed modularization can often help keep a system from becoming inscrutable by helping maintain expressive, scalable, testable systems that maintain production readiness.

Then, we'll put the microservice theory into practice by using Richard Rogers' microservice toolkit for Node, Seneca (https://github.com/rjrodger/seneca). Finally, we'll take a look at how to use Redis pub/sub as a cross-process communication system, thus demonstrating another way to compose your own microservice clusters.

Microservices
Any nontrivial, network-based application is composed of several independent subsystems that must cooperate to fulfill the business or other requirements of the larger system. For example, many web applications present browser-based interfaces composed of one or several libraries and/or UI frameworks translating user actions against JavaScript controllers into formalized network requests issued across several web protocols. These ultimately communicate with any number of servers running programs that implement various sorts of business logic—all sharing one or several databases, perhaps across several data centers. These initiate and coordinate even longer chains of requests.

Because there is no absolute *right way* to build software, every design is biased toward one or a few key principles, in particular, principles guiding how a system should scale, which normally affects how it is deployed. A few of the key principles informing the Node community—modular systems composed of small programs that do one thing well and are event-driven, I/O focused, and network focused—align closely with those underpinning microservices.
Microservice architecture designs typically respect the following principles:

- A system should be broken down into many small services that each do one thing and no more. This helps with clarity.
- The code-powering services should be short and simple. A common guideline in the Node community is to limit programs to somewhere near 100 lines of code. This helps with maintainability.
- No service should depend on the existence of another service or even know of the existence of other services. Services are decoupled. This helps with scalability, clarity, and maintainability.
- Data models should be decentralized, with a common (but not required) microservice pattern—that each service maintains its own database or a similar model. Services are stateless (this reinforces the previous point).
- Independent services are easy to replicate (or cull). Scaling (in both directions) is a natural feature of microservice architectures as new nodes can be added or removed as necessary. This also enables easy experimentation, where prototype services can be tested, new features can be tested or deployed temporarily, and so on.
- Independent, stateless services can be replaced or upgraded (or downgraded) independently regardless of the state of any system they form a part of. This opens the possibility of more focused, discrete deployments and refactors.
- Failure is unavoidable, so systems should be designed to fail gracefully. Localize points of failure (the first and second points of this list), isolate failure (the third and fourth points of this list), and implement recovery mechanisms (easier when error boundaries are clearly defined, small, and noncritical). Promote robustness by reducing the scope of unreliability.
- Testing is essential to any nontrivial system. Unambiguous and simple stateless services are easy to test. A key aspect of testing is simulation—the stubbing or mocking of services in order to test service interoperability. Clearly delineated services are also easy to simulate and can, therefore, be intelligently composed into testable systems.

The idea is simple: smaller services are easy to reason about individually, encouraging correctness of specifications (little or no gray area) and clarity of APIs (constrained sets of output follow constrained sets of input). Being stateless and decoupled, services promote system composability, help with scaling and maintainability, and are easier to deploy. Also, very precise, discrete monitoring of these sorts of systems is possible.
Redis pub/sub

In the previous chapter, we discussed the use of message queues, an excellent technique for rapid cross-process communication. Redis offers an interface allowing connected clients to subscribe to a particular channel and broadcast messages to that channel. This is generally described as a publish/subscribe paradigm. When you do not need more complex message exchanges and brokers but a simple and fast notification network, pub/sub works well.

Let's set up a basic pub/sub example and then move on to an example of using pub/sub to create a microservice architecture where many components doing a particular job are passed requests for their services and pass back results—all coordinated via Redis.

First, let's look at the most basic example of pub/sub—a script that demonstrates how to subscribe to a channel and how to publish to that channel:

```javascript
var redis = require("redis");

var publisher = redis.createClient();
var subscriber = redis.createClient();

subscriber.subscribe('channel5');

subscriber.on('message', function(channel, message) {
  console.log('channel: ', channel)
  console.log('message: ', message)
})

subscriber.on('subscribe', function() {
  publisher.publish('channel5', 'This is a message')
})
```

We are using Matt Ranney's Redis npm module. Find out more at https://github.com/mranney/node_redis.

To create both a publisher and a subscriber, we create two Redis clients. Note that, once a subscribe or psubscribe (more on psubscribe later) command is issued to a client, that client will enter subscriber mode, no longer accepting standard Redis commands. Typically, you will create two clients: one listening for messages on subscribed channels and the other a standard Redis client used for all other commands.
Also note that we must wait for the subscribe event to be fired on the subscriber client prior to publishing any messages. Redis does not hold a queue of published messages, which involves waiting for subscribers. A message for which there are no subscribers is simply dropped. The following is based on the Redis documentation:

"...published messages are characterized into channels, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more channels, and only receive messages that are of interest, without knowledge of what (if any) publishers there are. This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology."

So, we must wait for a subscriber prior to publishing. Once that subscription is made, we can publish to the channel5 channel, and the subscriber handle listening on that channel receives our message:

```
channel: channel5
message: This is a message
```

Let's take this a little further by creating two distinct Node processes, each performing a simple (micro) service. We'll build a calculator service with two operations—add and subtract. A separate, dedicated process will perform each operation, and the two-way communication between the calculator service and its helper services will be managed by Redis pub/sub.

First, we design two Node programs, one that adds and one that subtracts. We'll only show the adder here:

```javascript
var redis = require("redis");
var publisher = redis.createClient();
var subscriber = redis.createClient();

subscriber.subscribe('service:add');
subscriber.on('message', function(channel, operands) {
    var result = JSON.parse(operands).reduce(function(a, b) {
        return a + b;
    })
    publisher.publish('added', result);
})
subscriber.on('subscribe', function() {
    process.send('ok')
})
```

The subtraction program is nearly identical, differing only in the channel it listens on and the calculation it performs. These two services exist in the add.js and subtract.js files.
Managing Memory and Space

We can see what this service does. When it receives a message on the `service:add` channel, it will fetch the two operands passed to it, add them, and publish the result to the `added` channel. As we'll soon see, the calculator service will listen for results on the `added` channel. Also, you will notice a call to `process.send`—this is used to notify the calculator service that the add service is ready. This will make more sense shortly.

Now, let's build the `calculator.js` service itself:

```javascript
var redis = require("redis");
var publisher = redis.createClient();
var subscriber = redis.createClient();

var child_process = require('child_process');
var add = child_process.fork('add.js');
var subtract = child_process.fork('subtract.js');

add.on('message', function() {
  publisher.publish('service:add', JSON.stringify([7,3]))
})
subtract.on('message', function() {
  publisher.publish('service:subtract', JSON.stringify([7,3]))
})
subscriber.subscribe('result:added')
subscriber.subscribe('result:subtracted')
subscriber.on('message', function(operation, result) {
  console.log(operation + ' = ' + result);
});
```

The main calculator service forks two new processes running the `add.js` and `subtract.js` microservices. Typically, in a real system, the creation of these other services would be done independently, perhaps even on completely separate machines. This simplification is useful for our example, but it does demonstrate a simple way to create vertical scaling across cores. Clearly, each child process in Node on which `fork` has been used comes with a communication channel built in, allowing child processes to communicate with their parents as seen in the calculator service's use of `add.on(...)` and `subtract.on(...)` and in our calculation services with `process.send(...)`. 
Once the calculator service receives notice that its dependent services are ready, it publishes a request for work to be done on the service: add and service: subtract channels by passing operands. As we saw earlier, each service listens on its own channel and performs the work requested, publishing a result that this calculator service can then receive and use. When calculator.js is executed, the following will be displayed in your terminal:

```plaintext
result: subtracted = 4
result: added = 10
```

Earlier, we mentioned the psubscribe method. The p prefix signifies pattern and is useful when you want to subscribe to channels using a typical glob pattern. For example, rather than the calculator service subscribing to two channels with the common result: prefix, we can simplify it as follows:

```javascript
subscriber.psubscribe('result:*')
subscriber.on('pmessage', function(operation, result) {
    console.log(operation + ' = ', result);
})
```

Now, any additional service can publish results with the result: prefix and can be picked up by our calculator. Note that the p prefix must also be reflected in the pmessage event listener.

### Microservices with Seneca

Seneca is a Node-based microservice construction kit that helps you organize your code into distinct actions triggered by patterns. Seneca applications are composed of services that can accept JSON messages and, optionally, return some JSON. Services register an interest in messages with certain characteristics. For example, a service might run whenever a JSON message displaying `{ cmd: "doSomething" }` is broadcast.

To start, let's create a service that responds to two patterns, one pattern returning "Hello!" and the other returning "Goodbye!". Create a hellogoodbye.js file containing the following code:

```javascript
var seneca = require('seneca')();
var client = seneca.client(8080);

require('seneca')()
.add({'
    operation: 'sayHello'
},
    function(args, done) {

```
Managing Memory and Space

```javascript
done(null, {message: "Hello!"})
}
.add(
    operation:'sayGoodbye'
),
function(args, done) {
    done(null, {message: "Goodbye!"})
})
.listen(8080);

client.act({ operation: "sayHello" }, function(err, result) {
    console.log(result.message);
})

client.act({ operation: "sayGoodbye" }, function(err, result) {
    console.log(result.message);
})
```

The call to `seneca()` starts up a service that will listen on port 8080 on localhost for patterns rendered in the JSON format—one of either `{ operation: "sayHello" }` or `{ operation: "sayGoodbye" }`. We also create a client object connected to the Seneca service on 8080 and have that client act against those patterns. When this program is executed, you will see Hello! and Goodbye! displayed in your terminal.

Because the Seneca service is listening on HTTP by default, you can achieve the same result by making a direct call over HTTP, operating against the /act route:

```
curl -d '{"operation":"sayHello"}' http://localhost:8080/act
  // {"message":"Hello!"}
```

Now, let's replicate the calculator application developed earlier, this time using Seneca. We're going to create two services, each listening on a distinct port, with one performing addition and the other performing subtraction. As in the previous calculator example, each will be started as an individual process and called remotely.

Create an `add.js` file as follows:

```javascript
require('seneca')()
.add(
    operation:'add'
),
function(args, done) {
    var result = args.operands[0] + args.operands[1];
    done(null, {
        result : result
```
Next, create a subtract.js file identical to add.js, changing only its operation parameter and, of course, its algorithm:

```javascript
... 
.add({
   operation:'subtract'
 },
...
var result = args.operands[0] - args.operands[1];
... 
```

Open two terminals, and start both services:

```
node add.js
...
node subtract.js
```

To demonstrate the usage of these services, create a calculator.js file that binds a client to each service on its unique port and acts against them. Note that you must create distinct Seneca clients:

```
var add = require('seneca')().client({
   host:'127.0.0.1',
   port:8081
 });
var subtract = require('seneca')().client({
   host:'127.0.0.1',
   port:8082
 });
add.act({
   operation:'add',
   operands: [7,3]
 },
  function(err, op) {
    console.log(op.result)
  })
subtract.act({
   operation:'subtract',
   operands: [7,3]
});
```
Managing Memory and Space

```
function(err, op) {
    console.log(op.result)
}
```

Executing this program will result in the following:

```
10 // adding
4 // subtracting
```

Just as with the previous example, we can make a direct HTTP call:

```
curl -d '{"operation":"add","operands":[7,3]}' http://127.0.0.1:8081/act
// {"result":10}
```

By building out your calculator in this way, each operation can be isolated into its own service, and you can add or remove functionality as needed without affecting the overall program. Should a service develop bugs, you can fix and replace it without stopping the general calculator application. If one operation requires more powerful hardware or more memory, you can shift it to its own server without stopping the calculator application or altering your application logic—you only need to change the IP address of the targeted service. In the same way, it is easy to see how, by stringing together the database, authentication, transaction, mapping, and other services, they can be more easily modeled, deployed, scaled, monitored, and maintained than if they were all coupled to a centralized service manager.

Reducing memory usage

JavaScript was born and raised in the browser environment. For most of its history, this also implied that JavaScript programs were running on desktop systems with an enormous pool of available memory. For this reason, many JavaScript programmers have not traditionally thought much about managing memory in their applications.

In the world of Node, memory is not so cheap. According to Joyent (https://github.com/joyent/node/wiki/FAQ#what-is-the-memory-limit-on-a-node-process):

"Currently, by default, v8 has a memory limit of 512 MB on 32-bit systems and 1 GB on 64-bit systems. The limit can be raised by setting --max_old_space_size to a maximum of ~1024 (~1 GB) (32-bit) and ~1741 (~1.7 GiB) (64-bit), but it is recommended that you split your single process into several workers if you are hitting memory limits."

Let's go over possible strategies to reduce the amount of memory your Node programs consume. We'll end with a discussion of how to make use of two memory-efficient data structures supported by Redis when developing your projects.
Use streams, not buffers

The design and implementation of Node.js native modules follow a simple directive: keep everything asynchronous. This design ethic, by convention, informs the design of modules contributed by the Node community.

When a process operates synchronously, it holds, or locks, the total amount of memory it needs to fully complete, at which point the memory it has held is flushed, usually returning this result to the calling method or process. For example, the following operation will load the entirety of a file into the memory prior to returning it:

```javascript
var http = require('http')
var fs = require('fs')
http.createServer(function(req, res) {
  fs.readFile('./somefile.js', function(err, data) {
    res.writeHead(200);
    res.end(data)
  })
}).listen(8000)
```

When a request is made to localhost:8000, the somefile.js file is read off the filesystem in its entirety and returned to the client. That is the desired effect—but there is a slight problem. Because the entire file is being pushed into a buffer prior to being returned, an amount of memory equal to the byte size of the file must be allocated on each request. While the operation is itself asynchronous (allowing other operations to proceed), just a few requests for a very large file (of several MB, for example) can overflow the memory and take down the Node process.

Node excels at creating scalable web services. One of the reasons for this is the focus on providing robust Stream interfaces.

A better strategy is to stream the file directly to the HTTP response object (which is a writable stream):

```javascript
http.createServer(function(req, res) {
  fs.createReadStream('./static_buffered.js').pipe(res);
}).listen(8000)
```

In addition to requiring less code, data is sent (piped) directly to the out stream, using very little memory.
Managing Memory and Space

On the other hand, we can use Stream to enable a very nice and composable pipeline of transformations. There are several ways to achieve this goal (such as with Transform Stream), but we’ll just create our own transformer.

This script will take an input from process.stdin and convert what is received to uppercase, piping the result back to process.stdout:

```javascript
var Stream = require('stream')
var through = new Stream;
through.readable = true;
through.writable = true;
through.write = function(buf) {
  through.emit('data', buf.toString().toUpperCase())
}
through.end = function(buf) {
  arguments.length && through.write(buf)
  through.emit('end')
}
process.stdin.pipe(through).pipe(process.stdout);
```

As much as possible, convert your program logic into discrete stream transformations, and compose useful pipelines that do good things with data without touching the memory.

Understanding prototypes

JavaScript is an **Object-oriented (OO)** prototype-based language. It is important for you to understand what this means and how this sort of design is more memory efficient than many traditional OO language designs when used correctly. Because the storage of state data within Node processes is a common practice (such as connection data lookup tables within a socket server), we should leverage the prototypal nature of the language to minimize memory usage. What follows is a brief but pointed comparison of the classical inheritance-based object model and the object system that JavaScript provides in terms of memory usage and efficiency.

In class-based systems, a **class** contains instructions on how to create instances of itself. In other words, a class describes a set containing objects built according to a class specification, which includes things such as default values for attributes of constructed objects. To create an instance of a class, there must be a class definition that describes how to build that instance. Classes can also inherit properties from each other, creating new instance blueprints that share characteristics with other blueprints—an inheritance model describing the provenance of objects.
The primary purpose of any OO system is to facilitate the sharing of common knowledge between related objects. For example, this is how you would create two Point instances using an inheritance model:

![Diagram of Point Class and Instances]

Note that both instances now maintain an identical attribute structure. Additionally, the property x of both point instances has been copied from the base point class. Importantly, notice that the value of x has been copied to each instance even though this attribute value is identical in both instances.

Objects in a prototypal language do not require a class to define their composition. For example, an object in JavaScript can be created literally:

```javascript
var myPoint = {
  x: 100,
  y: 50
}
```

Not requiring the storage of a class definition prior to creating an object instance is already more memory efficient. Now, consider this use of prototypes to replicate the inheritance-based example discussed previously. In the following code, we see how a single object, myPoint, is passed as the first object to Object.create, which returns a new object with myPoint as its prototype:

```javascript
var myPoint = {
  x: 100,
  y: 50
}
var pointA = Object.create(myPoint, {
  y: 100
})
var pointB = Object.create(myPoint, {
  y: 200
})
```
Object.create is the preferred method in modern JavaScript (ES5+) to create objects. Older browsers will not support this syntax. For more information on compatibility, visit http://kangax.github.io/compat-table/es5/#Object.create.

This creates the following object construct:

![Diagram of object construction](image)

Note that each point instance does not store copies of attributes, the value of which is not explicitly declared. Prototypal systems employ message delegation, not inheritance. When a point instance receives the message *give me x*, and it cannot satisfy that request, it delegates the responsibility for satisfying that message to its prototype (which, in this case, does have a value for x). It should be obvious that, in real-world scenarios with large and complex objects, the ability to share default values across many instances without redundantly copying identical bytes will lead to a smaller memory footprint. Additionally, these instances can themselves function as prototypes for other objects, continuing a delegation chain indefinitely and enabling elegant object graphs using only as much memory as necessary to distinguish unique object properties.

Memory efficiency also speeds up instantiation. As should be clear from the preceding code, delegating responsibility for messages to a prototype implies that your extended receiver requires a smaller instance footprint—fewer slots need to be allocated per object. The following are two construction function definitions:

```javascript
var rec1 = function() {};
rec1.prototype.message = function() { ... }
var rec2 = function() {
    this.message = function() { ... }
}
```
Even with these simple definitions, many instances built from the first constructor will consume much less memory than an equal number of instances constructed from the second—new Rec1() will complete well before new Rec2() due to the redundant copying seen in the second prototype-less constructor.

You can see a performance comparison of the two instantiation methods at http://jsperf.com/prototype-speeds.

Use prototypes intelligently to reduce memory usage in your objects and to lower instantiation times. Determine the static or infrequently changed attributes and methods of your objects and put those into prototypes. This will allow you to create thousands of objects quickly, while reducing redundancy.

**Memory-efficient data structures with Redis**

While you should use the memory allotted to you in each Node process, more memory will likely be needed. In this section, we will look at Redis, an in-memory, high-speed database, and how it can be used to efficiently extend the amount of memory available to your programs.

At its most basic, Redis is a fast key-value store. We'll see later how it can be used as a cache for commonly used pieces of data. However, it also provides powerful data structures and an API allowing complex operations on those structures, thus helping with the modeling of sets of data and the relationships between sets of data. Here, we will discuss how to use Redis support for **Bit Operations (bitops)** and **HyperLogLog**—two space-efficient and, importantly, space-predictable memory structures to store and analyze the activity of data.

**Using bitwise operations to analyze user actions over time**

One of the more interesting features Redis provides is the ability to store binary numbers as values for keys. Multiple keys containing binary values can be compared by using the bitwise operators AND, OR, and XOR. By applying bitmasks mapping a range of bits to other binary values, you can make very rapid and memory-efficient analytical comparisons. In this section, we will learn some typical examples of how to use this technique.
Managing Memory and Space

Any key in a Redis database can store \((2^{32} - 1)\) bits or just under 512 MiB. This means that there are approximately 4.29 billion columns, or offsets, that can be set per key. This is a large number of data points referenced by a single key. We can set bits along these ranges to describe the characteristics of an item we would like to track, such as the number of users who have viewed a given article. Furthermore, we can use bit operations to gather other dimensions of information, such as what percentage of viewers of an article are female. Let's look at a few examples.

### Setting, getting, and counting bits

Let's assume that we are serving many different articles and each article is assigned a unique identifier. Also assume that we have 100,000 active members on our website, and that each user also has a unique identifier—a number between 1 and 100,000. Using bit operations, we can easily track article view activity on a given day by creating a key in Redis, which can be done by combining the article's unique key and a date string and setting bits at that key corresponding to the user ID associated with an article view. For example:

```
article:324:01-03-2014 : 00010100111010001001111...
```

This key represents article 324 on a specific date, efficiently storing the unique user IDs of viewers on that day by flipping a bit at an offset corresponding to the user's assigned ID. Whenever a user views an article, fetch that user's ID, use that number as an offset value, and use the `setbit` command to set a bit at that offset:

```
redis.setbit('article:324:01-03-2014', userId, 1)
```

In what follows, we're going to demonstrate how to use Redis bitops to efficiently store and analyze data. First, let's create data for three articles:

```javascript
var redis = require('redis');
var client = redis.createClient();
var multi = client.multi();
// Create three articles with randomized hits representing user views
var id = 100000;
while(id--) {
  multi.setbit('article1:today', id, Math.round(Math.random(1)));
  multi.setbit('article2:today', id, Math.round(Math.random(1)));
  multi.setbit('article3:today', id, Math.round(Math.random(1)));
}
multi.exec(function(err) {
  // done
});
```
Here, we simply created three Redis keys, 'article (1-3):today', and randomly set 100,000 bits on each key — either 0 or 1. Using the technique of storing user activity based on user ID offsets, we now have sample data for a hypothetical day of traffic against three articles.

We're using Matt Ranney's node_redis module (https://github.com/mranney), which supports the Redis multi construct, allowing the execution of several instructions in one pipeline rather than suffering the cost of calling each individually. Always use multi when performing several operations in order to speed up your operations. Note also how the ordering guarantees provided by Redis ensure ordered execution and how its atomicity guarantees that either all or none of the instructions in a transaction will succeed. See http://redis.io/topics/transactions.

To count the number of users who have viewed an article, we can use bitcount:

```javascript
client.bitcount('article1:today', function(err, count) {
  console.log(count)
})
```

This is straightforward: the number of users who saw the article equals the number of bits set on the key. Now, let's count the total number of article views:

```javascript
client.multi([
  ['bitcount', 'article1:today'],
  ['bitcount', 'article2:today'],
  ['bitcount', 'article3:today']
]).exec(function(err, totals) {
  var total = totals.reduce(function(prev, cur) {
    return prev + cur;
  }, 0);
  console.log("Total views: ", total);
})
```

Once multi returns an array of results corresponding to the results returned by Redis for each operation (a count of bits), we reduce the count to a sum representing the total number of views of all our articles.

If we are interested, instead, in how many articles user 123 has viewed today, we can use getbit, which simply returns the value (either 0 or 1) at a given offset. The result will be in the range 0–3:

```javascript
client.multi([
  ['getbit', 'article1:today', 123],
  ['getbit', 'article2:today', 123],
  ['getbit', 'article3:today', 123],
])
```
Managing Memory and Space

```javascript
//getbit, "article3:today", 123
].exec(function(err, hits) {
    var total = hits.reduce(function(prev, cur) {
        return prev + cur;
    }, 0);
    console.log(total); // 0, 1, 2 or 3
})
```

These are very useful and direct ways to glean information from bit representations. Let's go a little further and learn about filtering bits using bitmasks and the AND, OR, and XOR operators.

**Bitmasks and filtering results**

Previously, we learned how to count the number of articles user 123 has seen. What if we want to check whether user 123 has read both articles? Using the bitop AND, this is easy to accomplish:

```javascript
client.multi([
    ['setbit', 'user123', 123, 1],
    ['bitop', 'AND','123:sawboth','user123','article1:today',
     'article3:today'],
    ['getbit', '123:sawboth', 123]
]).exec(function(err, result) {
    var sawboth = result[2];
    console.log('123 saw both articles: ', !!sawboth);
});
```

First, we create a mask that isolates a specific user stored at the key 'user123', containing a single positive bit at offset 123 (again, representing the user's ID). The results of an AND operation on two or more bit representations is not returned as a value by Redis but rather written to a specified key, which is given in the preceding example as '123:sawboth'. This key contains the bit representation that answers the question whether both the article keys contain bit representations that also have a positive bit at the same offset as the 'user123' key.

What if we wanted to find the total number of users who have seen at least one article? The bitop OR works well in this case:

```javascript
client.multi([
    ['bitop', 'OR','atleastonearticle','article1:today',
     'article2:today','article3:today'],
    ['bitcount', 'atleastonearticle']
]).exec(function(err, results) {
    console.log("At least one: ", results[1]);
});
```

What if we wanted to find the total number of users who have seen at least one article? The bitop OR works well in this case.
Here, the 'atleastonearticle' key flags bits at all offsets that were set in any one of the three articles.

We can use these techniques to create a simple recommendation engine. For example, if we are able to determine via other means that two articles are similar (based on tags, keywords, and so on), we can find all users that have read one and recommended the other. To do this, we will use XOR in order to find all users that have read the first article or the second article, but not both. We then break that set into two lists: those who have read the first article and those who have read the second article. We can then use these lists to offer recommendations:

```javascript
client.multi([  
    ['bitop','XOR','recommendother','article1:today',  
    'article2:today'],  
    ['bitop','AND','recommend:article1','recommendother',  
    'article2:today'],  
    ['bitop','AND','recommend:article2','recommendother',  
    'article1:today'],  
    ['bitcount', 'recommendother'],  
    ['bitcount', 'recommend:article1'],  
    ['bitcount', 'recommend:article2'],  
    ['del', 'recommendother', 'recommend:article1',  
    'recommend:article2']  
]).exec(function(err, results) {  
  // Note result offset due to first 3 setup ops  
  console.log("Didn't see both articles: ", results[3]);  
  console.log("Saw article2; recommend article1: ", results[4]);  
  console.log("Saw article1; recommend article2: ", results[5]);  
});```

While it is not necessary, we also fetch a count of each list and delete the result keys when we are done.

The total number of bytes occupied by a binary value in Redis is calculated by dividing the largest offset by 8. This means that storing access data for even 1,000,000 users on one article requires 125 KB—not a lot. If you have 1,000 articles in your database, you can store full-access data for 1,000,000 users in 125 MB—again, not a very large amount of memory or storage to spend in return for such a rich set of analytics data. Also, the amount of storage needed can be precisely calculated ahead of time.

View the code bundle for an example of building a like this page service, where we use a bookmarklet to trigger likes on any URL using bit operations to store the time at which each like occurs (offsetting by the current second on a given day).
Managing Memory and Space

Other useful ways to deploy bitwise ideas are easy to find. Consider that if we allocate 86,400 bits to a key (the number of seconds in a day) and set a bit corresponding to the current second in the day, whenever a particular action is performed (such as a login), we have spent $\frac{86400}{8} / 1000 = 10.8$ KB to store login data that can easily be filtered using bitmasks to deliver analytics data.

As an exercise, use bitmasks to demonstrate gender breakdown in article readership. Assume that we have stored two keys in Redis, one reflecting the user IDs identified as female and the other as male:

```
users:female  : 00100001011000000011110010101...
users:male    : 11011110100111111100001101010...
```

Using bit operations, we filter articles by gender.

**Using HyperLogLog to count unique anonymous visitors**

One of the most common things done with databases is storing and counting unique things. How many events of a certain type have occurred? How many tags have been created?

Consider the nearly ubiquitous task of every marketer: counting the number of unique visitors to a web page. Traditionally, counting is done in databases by writing a row of data or in logs by writing a line of text whenever a visitor lands on a page. Each unique visit increments the set length by one. These are simple and straightforward techniques.

However, there is a problem: what if the same person arrives at the same page more than once? Whenever the user *John* lands on a page, some work must be done to determine whether this is a first-time occurrence (record it), or a repeat occurrence (don't record it). And there is another problem: the entire sequence of bytes representing a unique identifier—typically a very long hash—must be stored. Each unique item adds to the total memory expended in keeping track of item counts of the cardinality of a set. As we can't know in advance how many unique hits will occur, we cannot know how much memory will be needed to store this potential activity; we are, therefore, exposed to the risk of our system being overwhelmed when one page or another becomes very popular, goes viral, and so on, overnight.
HyperLogLog is a probabilistic data structure that allows a nearly infinite number of unique items to be counted within a fixed memory allocation. As Salvatore Sanfilippo puts it at http://antirez.com/news/75:

"HyperLogLog is remarkable as it provides a very good approximation of the cardinality of a set even using a very small amount of memory. In the Redis implementation it only uses 12kbytes per key to count with a standard error of 0.81%, and there is no limit to the number of items you can count, unless you approach 2^64 items (which seems quite unlikely)."

In your code bundle, you will find the /hyperloglog folder containing a simple counting application. Start this application by running server.js, and then, in your browser, visit localhost:8080. When you get there, click on the Send a specific value button. You should see the following output:

```
YOUR FINGERPRINT
RANDOM FINGERPRINT
GET UNIQUE COUNT
0
```

You have inserted the value 123 into a HyperLogLog key, and the number returned (1) is the cardinality of that key's set. Click on the same button a few times—given that this structure maintains a count of unique values, the number should not change. Now, try adding random values. You will see the numbers returned go up. Regardless of how many entries you make in the log key, the same amount of memory will be used. This sort of predictability is great when scaling out your application.

You can find the index.html page describing this client interface in the code bundle. All that the client needs to do is send an XHR request to localhost:8080/log/<some value>. Feel free to browse the code. More to the point, let's look at how the relevant route handler is defined on the server to insert values into HyperLogLog and retrieve log cardinality:

```javascript
var http = require('http');
var redis = require('redis');
var client = redis.createClient();
var hyperLLKey = 'hyper:uniques';

...

http.createServer(function(request, response) {
    var route = request.url;
```
Managing Memory and Space

```javascript
var val = route.match(/\^/log\/(.*)/);

if(val) {
    val = val[1];
    return client.pfadd(hyperLLKey, val, function() {
        client.pfcount(hyperLLKey, function(err, card) {
            respond(response, 200, JSON.stringify({
                count: err ? 0 : card
            }));
        });
    });
}).listen(8080)
```

After validating that we have received a new value on the `/log` route, we add that value to `hyperLLKey` using the `PFADD` command (in Redis, if a key does not exist when performing an insert operation, it is automatically created). Once inserted successfully, the key is queried for its `PFCOUNT`, and the updated set's cardinality is returned to the client.

In addition, the `PFMERGE` command lets you merge (create the union of) several HyperLogLog sets and fetch the cardinality of the resulting set. The following code will result in a cardinality value of 10:

```javascript
var redis = require('redis');
var client= redis.createClient();
var multi = client.multi();

client.multi(["pfadd", 'merge1', 1, 2, 3, 4, 5, 6, 10],
    ['pfadd', 'merge2', 1, 2, 3, 4, 5, 6, 7, 8, 9],
    ['pfmerge', 'merged', 'merge1', 'merge2'],
    ['pfcount', 'merged'],
    ['del', 'merge1', 'merge2', 'merged'])
    .exec(function(err, result) {
        console.log('Union set cardinality', result[3]);
    });
```

The ability to approximate the cardinality of merged sets brings to mind the sort of efficient analytics possibilities we saw when exploring bitwise operations. Consider HyperLogLog when counts of many unique values are useful analytically and an imprecise but very closely approximated count is sufficient (such as tracking the number of users who logged in today, the total number of pages viewed, and so on).
Taming V8 and optimizing performance

V8 manages Node's main process thread. When executing JavaScript, V8 does so in its own process, and its internal behavior is not controlled by Node. However, we can write JavaScript in a way that helps V8 achieve optimal compilation results. In this section, we'll focus on how to write efficient JavaScript and take a look at special configuration flags we can pass to V8 that help with keeping our Node process fast and light.

```
node -e "console.log(process.versions.v8)"
```

Optimizing JavaScript

The convenience of a dynamic language is in avoiding the strictness that compiled languages impose. For example, you need not explicitly define object property types and can actually change those property types at will. This dynamism makes traditional compilation impossible but opens up interesting new opportunities for exploratory languages, such as JavaScript. Nevertheless, dynamism introduces a significant penalty in terms of execution speeds when compared to statically compiled languages. The limited speed of JavaScript has regularly been identified as one of its major weaknesses.

V8 attempts to achieve the sorts of speeds with JavaScript that one observes for compiled languages. V8 attempts to compile JavaScript into native machine code rather than interpreting bytecode or using other just-in-time techniques. Because the precise runtime topology of a JavaScript program cannot be known ahead of time (the language is dynamic), compilation consists of a two-stage, speculative approach:

1. Initially, a first-pass compiler converts your code into a runnable state as quickly as possible. During this step, type analysis and other detailed analysis of the code is deferred, achieving fast compilation—your JavaScript can begin executing as close to instantly as possible. Further optimizations are accomplished during the second step.
2. Once the program is up and running, an optimizing compiler then begins its job of watching how your program runs and attempting to determine its current and future runtime characteristics, optimizing and re-optimizing as necessary. For example, if a certain function is called many thousands of times with similar arguments of a consistent type, V8 recompiles that function with optimized code. While the first compile step was conservative with an as-yet unknown and untyped functional signature, this hot function's predictable texture impels V8 to assume a certain optimal profile and recompile based on that assumption.

Assumptions help us make decisions more quickly but can lead to mistakes. What if the hot function that V8's compiler just optimized against a certain type signature is now called with arguments violating that optimized profile? V8 has no choice in that case: it must de-optimize the function — V8 must admit its mistake and roll back the work it has done. It will re-optimize in the future if a new pattern is seen. However, if V8 must again de-optimize at a later time and if this binary switching of optimizing/de-optimizing continues, V8 simply gives up and leaves your code in a de-optimized state.

Two areas of focus for the V8 team are achieving fast property access and dynamically creating efficient machine code. Let's look at ways to approach the design and declaration of arrays, objects, and functions so that you are helping, rather than hindering, the compiler.

**Numbers and tracing optimization/de-optimization**

The ECMA-262 specification defines the `Number` value as a *primitive value corresponding to a double-precision, 64-bit binary format IEEE 754 value*. The point is that there is no integer type in JavaScript; there is a `Number` type defined as a double-precision floating-point number.

V8 uses 32-bit numbers for all values internally for performance reasons that are too technical to discuss here. It can be said that, should greater width be needed, one bit is used to point to another 32-bit number. Regardless, it is clear that there are two types of values tagged as numbers by V8 and switching between these types will cost you something. Try to restrict your needs to 31-bit signed integers where possible.

Because of the type ambiguity of JavaScript, switching the types of numbers assigned to a slot is allowed. The following code does not throw an error:

```javascript
var a = 7;
a = 7.77;
```
However, a speculative compiler such as V8 will be unable to optimize this variable assignment given that its guess that a will always be an integer turned out to be wrong, forcing de-optimization.

We can demonstrate this using powerful V8 options available to you when executing code: executing V8 native commands in your Node program and tracing how V8 optimizes/de-optimizes your code.

Consider the following Node program:

```javascript
var someFunc = function foo(){}
console.log(%FunctionGetName(someFunc));
```

If you try to run this normally, you receive an Unexpected Token error—the modulo (%) symbol cannot be used within an identifier name in JavaScript. What is this strange method with a % prefix? It is a V8 native command, and we can turn to the execution of these types of functions using the --allow-natives-syntax flag as follows:

```
node --allow-natives-syntax program.js
// foo
```

You can learn about the available native functions by browsing the V8 source at https://code.google.com/p/v8/source/browse/trunk/src/runtime.cc?r=22500, and searching for runtime_function.

Now, consider the following code, which uses native functions to assert information about the optimization status of the square function using the %OptimizeFunctionOnNextCall native method:

```javascript
var operand = 3;
function square() {
    return operand * operand;
}
// Make first pass to gather type information
square();
// Ask that the next call of #square trigger an optimization attempt;
// Call
%OptimizeFunctionOnNextCall(square);
square();
```

Create a file using the preceding code and execute it using the following command:

```
node --allow-natives-syntax --trace_opt --trace_deopt myfile.js
```
Managing Memory and Space

You will see something like the following output returned:

```
[deoptimize context: c39daf14679]
[optimizing: square / c39dafca921 - took 1.900, 0.851, 0.000 ms]
```

We can see that V8 has no problem optimizing the `square` function as the operand is declared once and never changed. Now, append the following lines to your file and run it again:

```javascript
%OptimizeFunctionOnNextCall(square);
operand = 3.01;
square();
```

On this execution, following the optimization report given earlier, you should now receive something like the following output:

```
**** DEOPT: square at bailout #2, address 0x0, frame size 8
[deoptimizing: begin 0x2493d0fca8d9 square @2]
...
[deoptimizing: end 0x2493d0fca8d9 square => node=3, pc=0x29edb8164b46, state=NO_REGS, alignment=no padding, took 0.033 ms]
[removing optimized code for: square]
```

This very expressive optimization report tells the story very clearly—the once-optimized `square` function was de-optimized following the change we made in one number’s type. You are encouraged to spend time writing code and to test it using these methods now and as you move through this section.

Objects and arrays

As we learned when investigating numbers, V8 works best when your code is predictable. The same holds true with arrays and objects. Nearly all of the following bad practices are bad for the simple reason that they create unpredictability.

Remember that, in JavaScript, an object and an array are very similar under the hood. We won’t be discussing those differences but only the important similarities, specifically in terms of how both these data constructs benefit from similar optimization techniques.
Avoid mixing types in arrays. It is always better to have a consistent data type, such as all integers or all strings. Also, avoid changing types in arrays or in property assignments after initialization, if possible. V8 creates blueprints of objects by creating hidden classes to track types, and, when those types change, the optimization blueprints will be destroyed and rebuilt—if you’re lucky. See the following link for more information:

https://developers.google.com/v8/design

Don’t create arrays with gaps, an example of which is shown as follows:

```javascript
var a = [];
a[2] = 'foo';
a[23] = 'bar';
```

Sparse arrays are bad for this reason: V8 can either use a very efficient linear storage strategy to store (and access) your array data, or it can use a hash table (which is much slower). If your array is sparse, V8 must choose the less efficient of the two. For the same reason, always start your arrays at the zero index. Also, don’t ever use delete to remove elements from an array. You are simply inserting an undefined value at that position, which is just another way of creating a sparse array. Similarly, be careful about populating an array with empty values—ensure that the external data you are pushing into an array is not incomplete.

Try not to pre-allocate large arrays—grow as you go. Similarly, do not pre-allocate an array and then exceed that size. You always want to avoid spooking V8 into turning your array into a hash table.

V8 creates a new hidden class whenever a new property is added to an object constructor. Try to avoid adding properties after an object is instantiated. Initialize all members in constructor functions in the same order. Same properties + same order = same object.

Remember that JavaScript is a dynamic language that allows object (and object prototype) modifications after instantiation. Since the shape and volume of an object can, therefore, be altered after the fact, how does V8 allocate memory for objects? It makes certain reasonable assumptions. After a set number of objects is instantiated from a given constructor (I believe 8 is the trigger number), the largest of these is assumed to be of the maximum size, and all further instances are allocated that amount of memory (and the initial objects are similarly resized). A total of 32 fast property slots is then allocated to each instance based on this assumed maximum size. Any extra properties are slotted into a (slower) overflow property array that can be resized to accommodate any further new properties.
Managing Memory and Space

With objects, just as with arrays, try as much as possible to define the shape of your data structures in a futureproof manner, with a set number of properties, types, and so on.

**Functions**

Functions are typically called often and should be one of your prime optimization focuses. Functions containing try-catch constructs are not optimizable, nor are functions containing other unpredictable constructs, such as `with` and `eval`. If, for some reason, your function is not optimizable, keep its use to a minimum.

A very common optimization error involves the use of polymorphic functions. Functions that accept variable function arguments will be de-optimized. Avoid polymorphic functions.

**Caching strategies**

*Caching*, generally, is the strategy of creating easily accessible intermediate versions of assets. When retrieving an asset is expensive—in terms of time, processor cycles, memory, and so on—you should consider caching that asset. For example, if a list of Canadian provinces must be fetched from your database each time a person from that country visits, it is a good idea to store that list in a static format, obviating the expensive operation of running a database query on each visit. A good caching strategy is essential to any web-based application that serves large numbers of rendered data views, be they HTML pages or JSON structures. Cached content can be served cheaply and quickly.

Whenever you deploy content that doesn't change often, you most likely want to cache your files. Two general types of static assets are commonly seen. Assets such as a company logo, existing as-is in a content folder, will change very rarely. Other assets do change more often but much less frequently than on every request of the asset. This second class encompasses such things as CSS style sheets, lists of user contacts, latest headlines, and so on. Creating a reliable and efficient caching system is a nontrivial problem:

> "There are only two hard things in Computer Science: cache invalidation and naming things."

– Phil Karlton
In this section, we'll look at two strategies to cache your application content. First, we'll look at using Redis as an in-memory key-value cache for regularly used JSON data, learning about the Redis key expiry and key scanning. Finally, we'll investigate how to manage your content using the CloudFlare content delivery network (CDN), in the process learning something about using Node to watch for file changes and then invalidating a CDN cache when change events are detected.

**Using Redis as a cache**

In the example session-store implemented earlier, cookie values were stored in Redis and matched against incoming values to provide simple session management. This sort of regular checking of small, in-memory values is a common pattern in multiuser environments, for which technologies such as memcached were developed.

Redis is fully capable of functioning as a similar in-memory caching system. Let's implement a simple caching layer using Redis that intelligently manages key association and expiry.

Because many types of information will be cached, it is a good idea to namespace your cache keys. We'll structure our cache library such that individual namespace-aware cache APIs can be instantiated:

```javascript
var redis = require('redis');
var util = require('util');
var Promise = require('bluebird');
var Cache = function(config) {
    config = config || {};
    this.prefix = config.prefix ? config.prefix + ':' : 'cache:';

    var port = config.port || 6379;
    var host = config.host || 'localhost';

    this.client = redis.createClient(port, host, config.options || {});

    config.auth && this.client.auth(config.auth);
};
```

Typically, our caching layer will be decoupled from any given server, so here we design a constructor that expects Redis's connection and authentication information. Note the prefix argument. To instantiate a cache instance, use the following code:

```javascript
var cache = new Cache({ prefix: 'articles:cache' });
```
Also note that we're going to implement the cache API using **Promises** via the [bluebird library](https://github.com/petkaantonov/bluebird).

Getting a cached value is straightforward:

```javascript
Cache.prototype.get = function(key) {
    key = this.prefix + key;
    var client = this.client;
    return new Promise(function(resolve, reject) {
        client.hgetall(key, function(err, result) {
            err ? reject() : resolve(result);
        });
    });
};
```

All cache keys will be implemented as Redis hashes, so a GET operation will involve calling `hmget` on a key. The Promises-powered API now enables the following easy-to-follow syntax:

```javascript
cache.get('sandro').then(function(val) {
    console.log('cached: ' + val);
}).catch() {
    console.log('unable to fetch value from cache');
}
```

Setting a value is simply a matter of passing an object:

```javascript
Cache.prototype.set = function(key, val, ttl) {
    var _this = this;
    var pkey = this.prefix + key;
    var client = this.client;
    var setArr = [];

    for(var k in val) {
        setArr[k] = val[k];
    }
    return new Promise(function(resolve, reject) {
        client.hmset(pkey, setArr, function(err) {
            err ? reject() : resolve();
        ttl && _this.expire(key, ttl);
    });
});
```
When `val` is received, we reflect its key-value map in the Redis hash stored at `key`. The optional third argument, `ttl`, allows a flag to be set in Redis to expire this key after a number of seconds, flagging it for deletion. The key bit of code in this `expire` is the following:

```javascript
client.expire(key, ttl, function(err, ok) { // ...flagged for removal }
```

For more information on Redis `expire`, visit http://redis.io/commands/expire.

The `remove` method is simply a `del` operation on the Redis keyspace, so there is no need to explain it here. More interesting is the implementation of the `clear` method to remove all keys with a given prefix from Redis:

```javascript
Cache.prototype.clear = function() {
    var prefixMatch = this.prefix + '*';
    var client   = this.client;
    return new Promise(function(resolve, reject) {
        var multi = client.multi();
        (function scanner(cursor) {
            client.scan([cursor, 'match', prefixMatch], function(err, scn) {
                if(err) {
                    return reject();
                } // Add new delete candidates
                multi.del(scn[1]);
                if(+scn[0] !== 0) {
                    return scanner(scn[0]);
                } // Delete candidates, then resolve.
                multi.exec(resolve);})
            })(0);
        });
    });
```

Note the `scan` method we are using to target and delete keys matching our cache prefix. Redis is designed for efficiency, and, as much as possible, its designers aim to avoid adding slow features. Unlike other databases, Redis has no advanced `find` method of searching its keyspace, with developers limited to `keys` and basic glob pattern matching. Because it’s common to have many millions of keys in a Redis keyspace, operations using `keys`, unavoidably or through sloppiness, can end up being punitively expensive because a long operation blocks other operations—transactions are atomic, and Redis is single-threaded.
Managing Memory and Space

The `scan` method allows you to fetch limited ranges of the keyspace in an iterative manner, enabling (nonblocking) asynchronous keyspace scanning. The scan object itself is stateless, passing only a cursor indicating whether there are further records to be fetched. Using this technique, we are able to clean out all keys prefixed with our target cache key (pattern: `this.prefix + '*`). On each scan iteration, we queue up any returned keys for deletion using the `multi.del` function, continuing until the scanner returns a zero value (indicating that all sought keys have been returned), at which point we delete all those keys in one command.

Tie these methods together:

```javascript
  cache.set('deploying', { foo: 'bar' })
    .then(function() {
      return cache.get('deploying');
    })
    .then(function(val) {
      console.log(val); // foo:bar
      return cache.clear();
    })
    .then(cache.close.bind(cache));
```

This is a simple caching strategy to get you started. While managing key expiration yourself is a perfectly valid technique, as you move into larger production implementations, consider configuring Redis's eviction policies directly. For example, you will want to set the `maxmemory` value in `redis.conf` to some maximum upper bound for the cache memory and configure Redis to use one of the six documented eviction policies when memory limits are reached, such as **Least Recently Used (LRU)**. For more information, visit: [http://redis.io/topics/lru-cache](http://redis.io/topics/lru-cache).

Deploying CloudFlare as a CDN

A CDN is typically a globe-spanning network of servers leased out to companies unable to fund and build their own network. A CDN is set up to ensure that your application or other content remains available to anyone who wishes to access it, wherever they choose to access it in the world, and that your content is delivered quickly. Akamai is perhaps the most famous CDN, and CloudFlare is a recent arrival with a particular focus on security and "attack proofing" networks.

Usefully for our purposes, CloudFlare provides a free tier of service that satisfies the needs of most deployed applications. In the example that follows, you'll learn how to enable caching with CloudFlare. We'll then use the `cloudflare` module to purge your domain files when they change, in the process learning how to use Node's `fs.watch` method to watch for file changes.
CloudFlare has also embarked on an ambitious effort to host all the JS on its CDN at https://cdnjs.com/. Unlike other popular hosting services that only host the most popular JavaScript libraries, CloudFlare hosts all projects represented in the open GitHub project at https://github.com/cdnjs/cdnjs. Consider deploying your JavaScript files via this service.

To start, visit https://www.cloudflare.com/sign-up and set up a free account. You will need a domain to host files on—follow the instructions to configure your name servers and other DNS information. Once signed up, you will receive an authentication token and will use this to add CDN support to your application. CloudFlare does not cache HTML files by default. To enable HTML caching, visit your dashboard, locate your domain, open the options menu, and select Page rules. If your domain is foo.com, the following page rule will enable full caching: *foo.com/*.

Finally, locate the Custom Caching dropdown on the page rules admin page and select Cache everything.

Now, let's establish a connection with CloudFlare:

```javascript
var http = require('http');
var fs = require('fs');
var cloudflare = require('cloudflare');
var config = {
  "token": "your token",
  "email": "your account email",
  "domain": "yourdomain.com",
  "subdomain": "www",
  "protocol": "http"
};
var cloudflareClient = cloudflare.createClient({
  email: config.email,
  token: config.token
});
```

In our example, we will serve (and modify) a single index.html file. For this example, we will create a simple server:

```javascript
var indexFile = './index.html';
http.createServer(function(request, response) {
  var route = request.url;
  if(route === "/index.html") {
    response.writeHead(200, {
      "content-type": "text/html",
      "cache-control": "max-age=31536000"
    });
    response.end(fs.readFileSync(indexFile));
  } else {
    response.writeHead(404);
  }
});
```

CloudFlare has also embarked on an ambitious effort to host all the JS on its CDN at https://cdnjs.com/. Unlike other popular hosting services that only host the most popular JavaScript libraries, CloudFlare hosts all projects represented in the open GitHub project at https://github.com/cdnjs/cdnjs. Consider deploying your JavaScript files via this service.
Note how max-age is set on the cache-control header. This will indicate to CloudFlare that we want this file cached.

With the server set up, we will now add the following purge method:

```javascript
function purge(filePath, cb) {
  var head = config.protocol + '://';
  var tail = config.domain + '/' + filePath;
  // foo.com && www.foo.com each get a purge call
  var purgeFiles = [
    head + tail,
    head + config.subdomain + '.' + tail
  ];
  var purgeTrack = 2;
  purgeFiles.forEach(function(pf) {
    cloudflareClient.zoneFilePurge(config.domain, pf,
    function(err) {
      (--purgeTrack === 0) && cb();
    });
  });
}
```

When this method is passed a file path, it asks CloudFlare to purge its cache of this file. Note how we must use two purge actions to accommodate subdomains.

With purging set up, all that is left to do is watch the filesystem for changes. This can be accomplished via the fs.watch command:

```javascript
fs.watch('./index.html', function(event, filename) {
  if(event === "change") {
    purge(filename, function(err) {
      console.log("file purged");
    });
  }
});
```
Now, whenever the index.html file is changed, our CDN will flush its cached version. Create that file, start up the server, and point your browser to localhost:8080, bringing up your index file. In your browser's developer console, inspect the response headers—you should see a CF-Cache-Status: MISS record. This means that CloudFlare (CF) has fetched and served the original file from your server—on the first call, there is no cached version yet, so the cache was missed. Reload the page. The same response header should now read CF-Cache-Status: HIT. Your file is cached!

Go ahead and change the index file in some way. When you reload your browser, the changed version will be displayed—its cached version has been purged, the file has been fetched once again from your server, and you will see the MISS header value again.

You will want to expand this functionality to include a larger group of files and folders. To learn more about fs.watch, go to http://nodejs.org/api/fs.html#fs_fs_watch_filename_options_listener.

Managing sessions

The HTTP protocol is stateless. Any given request has no information about previous requests. For a server, this means that determining whether two requests originated from the same browser is not possible without further work. That's fine for general information, but targeted interactions require a user to be verified via some sort of unique identifier. A uniquely identified client can then be served targeted content—from lists of friends to advertisements.

This semipermanent communication between a client (often a browser) and a server persists for a period of time—at least until the client disconnects. That period of time is understood as a session. An application that manages sessions must be able to create a unique user session identifier, track the activity of an identified user during that session, and disconnect that user when requested or for some other reason, such as on reaching a session limit.

In this section, we'll implement a JSON Web Token (JWT) system for session management. JWT's have an advantage over traditional cookie-based sessions in that they do not require the server to maintain a session store as JWTs are self-contained. This greatly helps with deployments and scaling. They are also mobile friendly and can be shared between clients. While a new standard, JWTs should be considered as a simple and scalable session storage solution for your applications.
Managing Memory and Space

**JSON Web Token authentication and sessions**

A basic authentication system might require a client to send a username and password on each request. To initiate a token-based authenticated session, a client sends credentials just once, receives a token in exchange, and then sends only that token on subsequent requests, gaining any access that token provides. Incessantly passing around sensitive credentials is no longer required, as the following diagram demonstrates:

One particular advantage of JWTs is that servers are no longer responsible for maintaining access to a common database of credentials as only the issuing authority needs to validate an initial sign-in. There is no need to maintain a session store when you are using JWTs. The issued token (think of it as an access card) can, therefore, be used within any domain (or server) that recognizes and accepts it. In terms of performance, the cost of a request is now the cost of decrypting a hash versus the cost of making a database call to validate credentials. We also avoid the problems we can face using cookies on mobile devices, such as cross-domain issues (cookies are domain-bound), certain types of request forgery attacks, and so on.

Let’s look at the structure of a JWT and build a simple example demonstrating how to issue, validate, and otherwise use JWTs to manage sessions.

A JWT token has the following format:

```
<base64-encoded header>.<base64-encoded claims>.<base64-encoded signature>
```

Each segment is described in the JSON format.
A **header** simply describes the token—its type and encryption algorithm. Take the following code as an example:

```json
{
    "typ": "JWT",
    "alg": "HS256"
}
```

Here, we declare that this is a JWT token, which is encrypted using **HMAC SHA-256**.

See [http://nodejs.org/api/crypto.html](http://nodejs.org/api/crypto.html) for more information about encryption and how to perform encryption with Node. The JWT specification itself can be found at [http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html](http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html). Note that the JWT specification is in a draft state at the time of writing this, so changes may be made in the future.

The **claims** segment outlines security and other constraints that should be checked by any service receiving the JWT. Check the specification for a full accounting. Typically, a JWT claims manifest will want to indicate when the JWT was issued, who issued it, when it expires, who the subject of the JWT is, and who should accept the JWT:

```json
{
    "iss": "http://blogengine.com",
    "aud": ["http://blogsearch.com", "http://blogstorage"],
    "sub": "blogengine:uniqueuserid",
    "iat": "1415918312",
    "exp": "1416523112",
    "sessionData": "<some data encrypted with secret>"
}
```

The **iat** (issued-at) and **exp** (expires) claims are both set to numeric values indicating the number of seconds since the Unix epoch. The **iss** (issuer) should be a URL describing the issuer of the JWT. Any service that receives a JWT must inspect the **aud** (audience), and that service must reject the JWT if it does not appear in the audience list. The **sub** (subject) of the JWT identifies the subject of the JWT, such as the user of an application—a unique value that is never reassigned, such as the name of the issuing service and a unique user ID.

Finally, useful data is attached using a key-value pairing of your choice. Here, let's call the token data **sessionData**. Note that we need to encrypt this data—the signature segment of a JWT prevents tampering with session data, but JWTs are not themselves encrypted (you can always encrypt the entire token itself though).
Managing Memory and Space

The last step is to create a signature, which, as mentioned, prevents tampering—a JWT validator specifically checks for mismatches between the signature and the packet received.

What follows is a scaffold server and client example demonstrating how to implement a JWT-driven authentication system. Rather than implementing the various signing and validation steps by hand, we’ll use the jwt-simple package. Feel free to browse the /jwt folder in your code bundle, which contains the full code we’ll be unpacking next.

To ask for a token, we will use the following client code:

```javascript
var token;

function send(route, formData, cb) {
  if(!(formData instanceof FormData)) {
    cb = formData;
    formData = new FormData();
  }
  var caller = new XMLHttpRequest();
  caller.onload = function() {
    cb(JSON.parse(this.responseText));
  };
  caller.open("POST", route);
  token && caller.setRequestHeader('Authorization', 'Bearer ' + token);
  caller.send(formData);
}
// ...When we have received a username and password in some way
formData = new FormData();
formData.append("username", username);
formData.append("password", password);
send("/login", formData, function(response) {
  token = response.token;
  console.log('Set token: ' + token);
});
```

We’ll implement the server code next. For now, note that we have a send method that expects, at some point, to have a global token set for it to pass along when making requests. The initial /login is where we ask for that token.
Using the Express web framework, we create the following server and `/login` route:

```javascript
var express = require('express');
...
var jwt = require('jwt-simple');
var app = express();

app.set('jwtSecret', 'shhhhhhhhhhh');

app.post('/login', auth, function(req, res) {
  var nowSeconds   = Math.floor(Date.now()/1000);
  var plus7Days   = nowSeconds + (60 * 60 * 24 * 7);
  var token = jwt.encode({
    "iss": "http://blogengine.com",
    "aud": ["http://blogsearch.com", "http://blogstorage"],
    "sub": "blogengine:uniqueuserid",
    "iat": nowSeconds,
    "exp": plus7Days
  }, app.get('jwtSecret'));

  res.send({
    token : token
  });
});
```

Note that we store `jwtSecret` on the app server. This is the key that is used when we are signing tokens. When a login attempt is made, the server will return the result of `jwt.encode`, which encodes the JWT claims discussed previously. That's it. From now on, any client that mentions this token to the correct audience will be allowed to interact with any services those audience members provide for a period expiring 7 days from the date of issue. These services will implement something like the following code:

```javascript
app.post('/somesservice', function(req, res) {
  var token = req.get('Authorization').replace('Bearer ', '');
  var decoded = jwt.decode(token, app.get('jwtSecret'));
  var now = Math.floor(Date.now()/1000);
  if(now > decoded.exp) {
    return res.end(JSON.stringify({
      error : "Token expired"
    }));
  }
  res.send(<some sort of result>);
});
```
Managing Memory and Space

Here, we are simply fetching the Authorization header (stripping out Bearer) and decoding via jwt.decode. A service must at least check for token expiry, which we do here by comparing the current number of seconds from the epoch to the token's expiry time.

Using this simple framework, you can create an easily scalable authentication/session system using a secure standard. No longer required to maintain a connection to a common credentials database, individual services (deployed perhaps as microservices) can use JWTs to validate requests, incurring little CPU latency or memory cost.

Summary

We covered a lot of ground in this chapter. Best practices for writing efficient JavaScript that the V8 interpreter can handle properly were outlined, including an exploration of garbage collection, the advantages of Node streams, and how JavaScript prototypes should be deployed in order to save memory. Continuing with the theme of reducing storage costs, we explored various ways in which Redis can help with storing large amounts of data in a space-efficient way.

Additionally, we looked at strategies to build composable, distributed systems. In the discussion on microservices, we touched on approaches to network many individual services and build the networks they can use to communicate with each other—from pub/sub to Seneca’s pattern and action models. Joined with the examples of caching techniques, a reasonably complete picture of the issues you might want to consider when planning out resource management in your application was established.

After you build up a reasonably sophisticated architecture, it becomes more and more necessary to build probes and other monitoring tools to stay on top of what is going on. In the next chapter, we’ll build tools to help you trace the changing topology of running applications.
Where to buy this book

You can buy Deploying Node.js from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.